NASA-CR-194043

21 p

Experimental Facility for Implementing Distributed Database Services in Operating Systems*

Bharat Bhargava
Enrique Mafla
John Riedl

Technical Report Number CSD-TR-930

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract

Distributed database systems need special operating system support. Support routines can be implemented inside the kernel or at the user level. The decision depends on the tradeoff between performance and complexity. Kernel-level functions while efficient, are hard to implement. User-level implementations are generally penalized by poor performance and lack of security. This paper proposes a new approach to supplement and/or modify kernel facilities for database transaction processing. Our experimental facility called Push is based on an extension language interpreted within the kernel, that provides the flexibility and security required. Our implementation provides the efficiency of kernel-resident code as well as the simplicity and safety of user-level programming. This facility enables experimentation that would be difficult and time-consuming in current environments. The overhead of the Push implementation can be factored out to give a good estimate of the performance of a native kernel implementation. We have used Push to implement several kernel-resident services. A multicast implementation in Push has an inherent overhead of 0.32 milliseconds per additional site. The corresponding overhead for direct kernel-level implementation is 0.15 milliseconds and for a user-level implementation 0.57 milliseconds.

*This research is supported by NASA and AIRMICS under grant number NAG-1-676, NSF grant IRI-8821398, and AT&T.

1 Introduction

Operating system services have to be constantly modified and extended in order to adjust the system to changing environments and applications. New or alternative operating system facilities can be implemented either inside the kernel or in user-level processes. Many times, the decision is based on the simplicity versus efficiency argument. Complexity and efficiency are characteristic of kernel-resident code, while simplicity and poor performance are characteristic of user-level code. This paper describes a system called *Push*, that facilitates changing the functionality of the operating system kernel dynamically. It combines the flexibility and safety of user-level code with the efficiency and security of kernel-level code. Push can be used to implement semantically-rich system call interfaces that provide enhanced support for specific transaction processing systems.

The Push system consists of a *Push machine*, a *Push assembler*, and a set of *Push utilities*. The Push machine is incorporated in the operating system kernel. It allows users to run their own code inside the kernel. The Push machine hides the complex kernel data structures and mechanisms from the user. The interface offered by the Push machine is independent of the hardware and operating system. The assembler translates user-level code to the internal representation understood by the Push machine. Push utilities initialize the Push environment, add/delete assembled Push programs to/from the kernel, and print information about loaded Push programs. A prototype of this system has been implemented in the context of the Unix[1] operating system. We have used this prototype to conduct experiments on new kernel-resident support for distributed transaction processing.

There are two types of applications for Push. It can be used as an experimental tool or as an operational tool. As an experimental tool, Push can be used to prototype different alternatives that provide particular operating system services. The prototypes can then be tested in the target environment before making the final implementation in the kernel. Push has the advantage that the rest of the system is not disrupted while the experiments are taking place. There is no need to recompile and reboot the kernel. In addition, the protection scheme of Push avoids system crashes due to bugs in the new services. When Push is used as an operational tool, Push routines can be added to or deleted from the kernel dynamically during normal operation of the system. This feature introduces a form of adaptability to the system.

Database implementors have suggested that additional support in the underlying operating system is needed for efficiency [Sto81, SDE85, Spe86]. Push provides a facility for experimenting with new or extended operating system services. Examples of these services include buffer management, file system support, process management, interprocess communication, concurrency control, atomicity control, and crash recovery. The services that are present in current operating systems are general-purpose and do not satisfy the demands of

[1]Unix is a trademark of AT&T Bell Laboratories.

distributed transaction processing algorithms [Sto81, SDE85, BM89]. For instance, locking facilities and buffer management are generally implemented by database systems because the services provided in operating systems are inadequate.

Section 2 discusses design, implementation, and performance issues of Push. Section 3 describes experiments conducted with Push. In section 4, we describe alternative approaches that have been used to achieve flexible and adaptable operating systems. Finally, section 5 summarizes the paper and describes our future plans in this area.

2 Design and Implementation of Push

Push is a new approach for kernel extensibility. New algorithms can be coded in Push and loaded into the kernel without disturbing the rest of the system. We are specifically interested in kernel-resident services for efficient support of transaction processing. For instance, a multi-phase commit protocol can be written in this language that would send and receive two rounds of messages with a single system call.

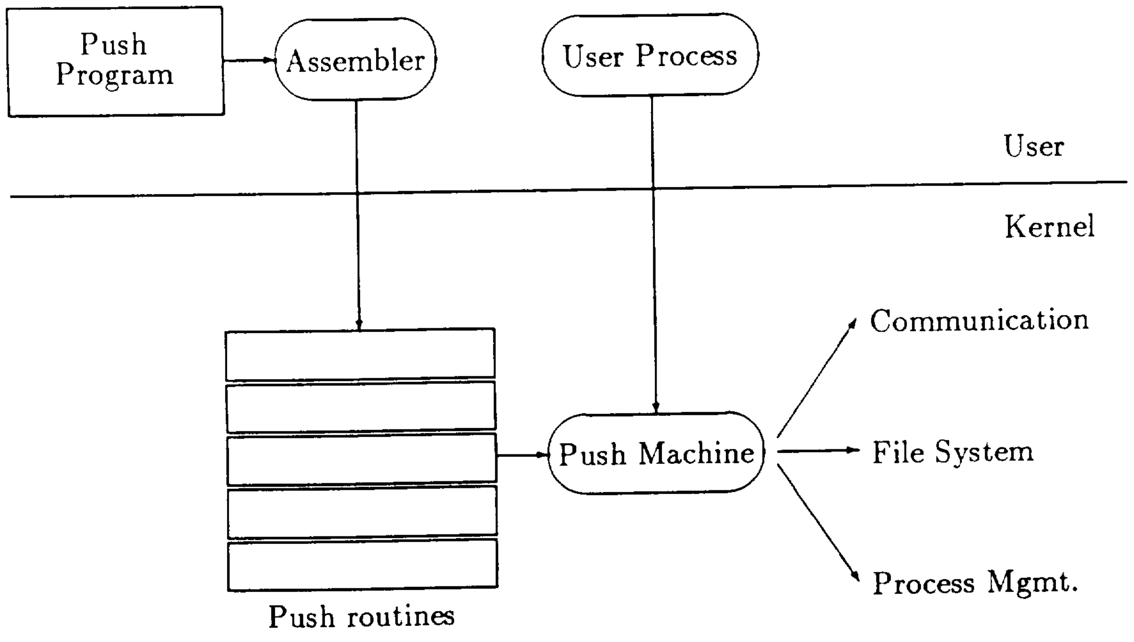

Figure 1: The Push system architecture

Figure 1 shows the details of the Push architecture. The user writes a desired service in a high-level language. The user program is assembled into Push machine code. This code is then loaded into the kernel and stored in a special data structure. Now, the user can use the new operating system feature by invoking the corresponding Push routine with a special system call. This system call activates the kernel-resident Push machine, which runs the Push program on behalf of the user. The Push virtual machine provides the user with a high-level abstraction of basic kernel services, including primitives for process management, file system services, and interprocess communication.

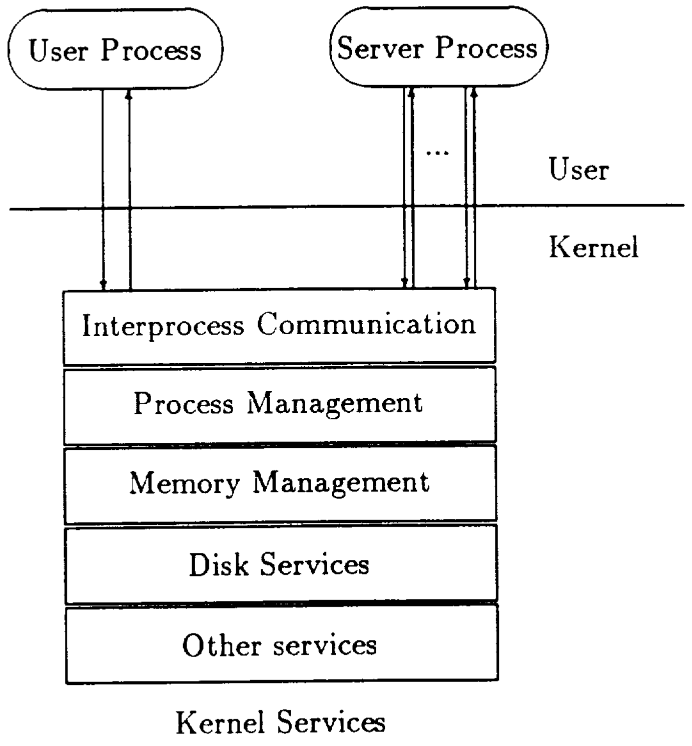

Figure 2: The server approach

Figure 2 illustrates the alternative approach of having the new service implemented at the user level, as a separate server process. Note the context switch overhead introduced by the frequent need to cross the user-kernel boundary. The boundary crossing is necessary for two reasons. The user process and the server can communicate only through the kernel. Moreover, the server needs to access kernel tables and routines via the system call interface. For example, if the server process implements multicasting, the number of user-kernel interactions grows proportional to the number of members in the destination multicast group. In contrast, the Push approach requires only one such interaction.

2.1 Design Issues

In designing Push there are several considerations.

1. The Push machine should protect the rest of the kernel address space from access by the Push programs. An erroneous program may produce incorrect results for its users, but it must not violate the integrity of the kernel.

2. Push programs must be efficient to execute. If Push is inherently slow, the primary goal of achieving high performance cannot be met.

3. Push should provide simple timer services to the programs. In a distributed environment, error handling must include support for detecting lost messages.

4. A Push program must not be able to monopolize the CPU.

There are several approaches to protect the kernel address space from arbitrary access by Push programs. The first is to develop a user-level compiler that produces type-safe code, compiling in run-time checks where necessary. The compiler would mark the programs in an unforgeable manner and a privileged loader would be the only program with permission to push programs into the kernel[2]. Alternatively, the kernel could accept programs in the high-level language and compile the programs itself. The difficulty with these two approaches is that such a compiler would be difficult to port to new architectures. In addition, the loading of compiled programs safely into the kernel would be tricky. Implementing a compiler in the kernel has the further disadvantage that it would increase the kernel size. We chose to design a virtual machine within the kernel for running user programs. The Push machine is stack-based, with a simple instruction set, and a design that provides for simple implementation.

Performance is a potential problem of the virtual machine approach. Both the size of the virtual machine and the execution time of the Push instructions must be kept low. The size of the Push machine will affect the space left for user processes, and may lead to increasing the paging activity in the system. In order to determine if favorable performance results can be achieved by the use of Push, we have to contrast the interpretation overhead with the disadvantages of user-level code.

In addition to protecting the kernel address space, we must prevent the monopolization of the CPU by the processes running Push programs. This protection is achieved by running the programs with interrupts enabled. While executing kernel routines such as 'receive', interrupts are disabled as usual, but Push has no command to affect the interrupt status. Hence clock interrupts will occur as usual, and the kernel will make its normal time-slicing decisions. Unfortunately, Unix only replaces the executing process upon entering or exiting the kernel, and Push programs may loop indefinitely within the kernel. Our solution is to add code that checks for runaway Push programs to the clock interrupt routine. If a Push program is running when a clock interrupt occurs, the routine increments a special 'wound' counter in the Push program. If the wound counter is incremented beyond a fixed limit, the interrupt routine terminates the Push program, returning an error message to the

[2]For instance, the compiler could include a cryptographic checksum in the compiled program.

user. In addition, the Push program is purged from the table of programs and a message is printed on the console, so that the same program does not continue to monopolize the CPU. Long-running Push programs may need a method to increase the number of clock ticks permitted.

Many of the Push programs will need timer services so messages can be retransmitted or timeout failures can be returned to the user. Our design supports a simple timeout facility that invokes the program at a specified label after a certain time (specified in milliseconds) elapses. The timeout is supported by the clock interrupt routine that keeps a list of pending timeouts in an increasing order of time. When a timeout expires, the clock routine checks to see if the program is still active. If so, the clock routine cleans up any queues on which the program was waiting, sets its execution point within the interpreter to the specified address, and returns the calling process to the run queue. When the process is rescheduled, it begins interpreting again at the new address.

2.2 Push Language Details

Push provides a simple stack-based language which can be executed efficiently within the kernel. The programs consist of two sections. The declaration section includes the declaration of input-output parameters, constants, and local variables. Parameters are of three types: input, output, and inout. Parameters and local variables can be defined as integers or as pointers to strings of bytes. Push programs may invoke the kernel memory allocator to initialize pointers. The executable section consists of a sequence of Push instructions. In addition to the stack operations, Push provides special operations that allows the user to access basic kernel services. Appendix A summarizes the operations available in Push. One operation is specified per line. Labels, if present, must proceed the operation code and the operands. Comments preceded by the character % can be inserted in a separate line or after a Push statement. Appendix B shows a sample Push program that implements multicasting.

The current implementation of the Push system includes an assembler for the stack language. The assembler translates user-level programs into Push machine code. This code is represented as an array of 4-byte words. Each declaration and instruction in the program is represented by one such word. The first byte stores the operation code, the second byte encodes information about the nature of the operand, and the last two bytes are used to store the operand itself. The operand can be a constant, a Push variable, or a pointer to a Push variable. The assembler is 884 lines of C code, and compiles to 60 Kbytes, unoptimized. A Push disassembler is 332 lines of C code, and 20 Kbytes compiled. A future implementation will include a compiler from a subset of C to the assembly language.

2.3 The Push Machine

Assembled Push programs are loaded into the kernel using a special system call, Pushcode. Pushcode takes two arguments: the name of a Push program and the address of the assembled program. The programs are stored in an array and are looked up by name when invoked. A table keeps information about the Push programs loaded into the kernel. This information includes the name of the program, its kernel address, length, owner, and access rights. The owner of a program can execute, remove, and overwrite it. Programs can be marked as sharable. This means that other users beside the owner can execute it. Program names that will be used by several users should be registered before users are permitted to login and marked as sharable. A separate system call is used to remove a program from the kernel's table. A third system call prints information about the loaded programs. A shell[3] program accepts the name of a source Push routine, assembles it, and loads the assembled code into the kernel using the Pushcode system call.

A Push procedure that has been loaded into the kernel is invoked by a special system call. Pushrun. The call to Pushrun requires two arguments: the name of the Push procedure to be invoked and a pointer to a vector of arguments for the Push procedure. Each executing Push program is provided with an execution stack which contains the parameters, local variables, and the values dynamically pushed into it while the program is running. When a procedure is invoked, the arguments indicated in the program definition as input or inout are copied into the kernel address space. Arguments indicated as output are copied from kernel to user address space immediately before the Push procedure returns. Push programs can allocate/deallocate memory dynamically. A table records the address, length, and read/write access rights of allocated memory. When a process wants to access a block of dynamic memory to read or write, Push checks the boundaries of that block of memory against the information kept in the table. When the program terminates, all allocated memory is released automatically.

The first implementation of Push runs inside SunOS 4.0 in Sun[4] 3/50's. The interpreter consists of 800 lines of C code, and takes about 10 Kbytes of memory. Ten Push programs of 100 statements each consume 5 Kbytes, including the run time stack. The entire Push implementation increases the size of the kernel by less than 20 Kbytes, which is relatively small compared to the total size of the kernel. We are using a streamlined version of SunOS 4.0, which is 584 Kbytes, including the Push interpreter.

[3]A shell is a Unix command-line interpreter.

[4]SunOS and Sun are trademarks of Sun Microsystems

3 Experiments with Communication and Distributed Commitment

In order to illustrate the utility of the Push software, we developed several database-oriented services. These services include the multicast routine listed in appendix B, a multi RPC facility, a distributed commitment protocol, and the file copy utility shown in appendix C. In this section, we compare the performance of the Push programs with the performance of similar services implemented at the kernel and user levels. To implement the user and kernel versions of the communication services, we used the SE suite of protocols. SE (*S*imple *E*thernet) is a set of streamlined, low overhead communication protocols for the Ethernet [BMR87]. The three services compared in each of these experiments provide the same functionality.

Experimental Method. All of the experiments were run in similar conditions. The machines were idle, and the measurements were taken at night when the network was relatively idle. The timings were done on a Sun 3/50 with a special microsecond resolution clock[5]. The file system and multicasting experiments were done with the system in single-user mode, and physically disconnected from the rest of the ethernet. Confidence intervals computed for the multicasting experiment were always less then 5% at 95% confidence. Confidence intervals for the other experiments have not yet been computed, but are also expected to be good.

3.1 Multicasting

The programs considered for this experiment send a 20 byte message to the set of destinations in the multicast group and return. The user-level SE multicast utility is implemented on top of the SE device driver, which provides point to point Ethernet communication. In order to support multicast, this utility has to call the device driver for each member in the multicast group. The kernel-level SE multicast utility uses the *multiSE* device driver [BMRS89]. This device driver can send the same message to a group of destinations on the Ethernet with one system call. Figure 3 shows these three approaches for multicasting.

The simulation of multicast inside the kernel is an important service for short-lived multicast groups. Short-lived multicast groups are frequently used in distributed transaction processing systems. Each transaction involves a different subset of sites, based on the distribution of replicas of items read or written[BHG87]. Multicasting to the subset of sites happens during transaction processing (to read/write or to form quorums[Gif79]) and during transaction commitment. There are too many such subsets to define multicast groups for each possible subset. The use of multicast mechanisms that require group initialization is inadequate, because of the overhead of setting up the multicast group.

[5]The times were collected using Peter Danzig's and Steve Melvin's timer board. It uses the timer chip AM9513A from Advanced Micro Devices, Inc. The timer has a resolution of up to four ticks per microsecond.

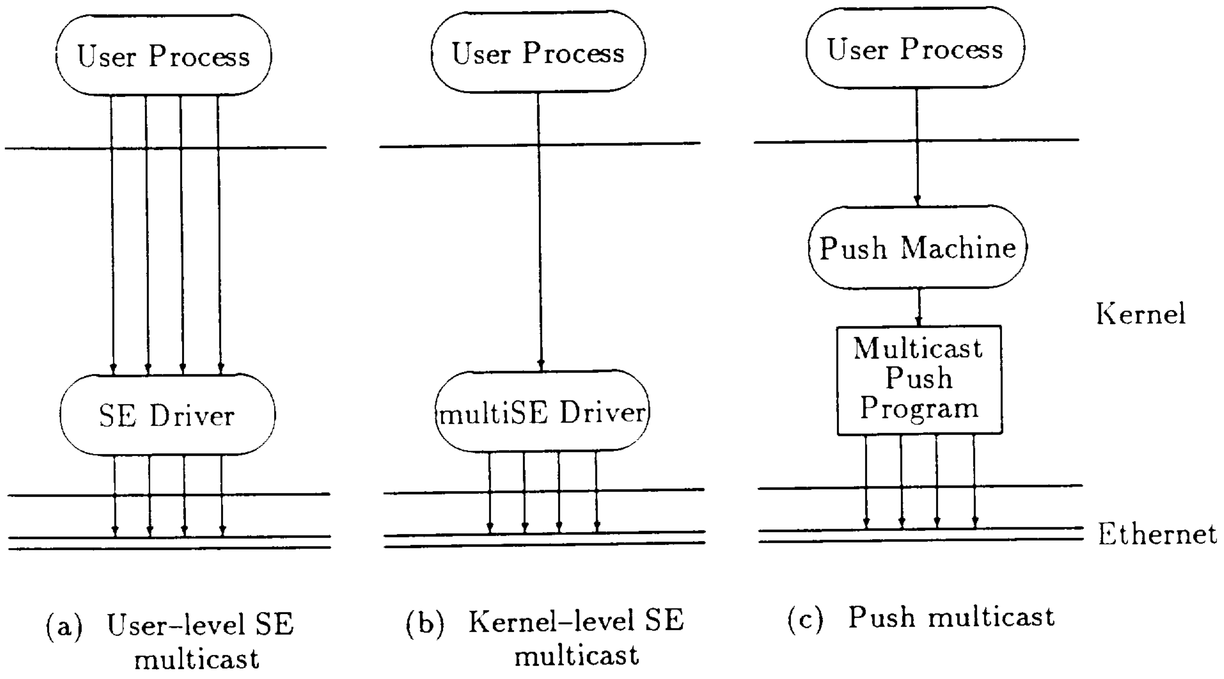

Figure 3: Approaches for Multicasting

In table 1, we compare the performance of the three multicast methods. Kernel-level SE multicast shows the best performance, and user-level SE multicast the worst. The difference between the times for kernel-level SE and Push is due to the interpretation overhead of the Push program. On the other hand, the *multiSE* driver takes significantly more effort to implement, debug and maintain.

A more precise picture of the intrinsic preformance of the three methods is presented in table 2. The table shows the overhead added per additional destination in the multicasting group. This overhead includes the time consumed by the network interface, which is fixed. In our case, this time (0.6 ms) includes the conversion of the message to *mbufs*[6], their transmission over the cable, and the processing of the corresponding interrupt. The first column represents the net overhead of each method. The execution of the loop in the Push program (13 Push instructions) takes about 320 μs, which averages 25 μs per instruction.

[6]Mbufs are special buffers used by the Unix communication subsystem.

Number of destinations	kernel level SE	user level SE	Push
1	1.2	1.2	2.7
5	4.2	5.9	6.6
10	8.0	11.7	11.0
15	11.7	17.5	15.6
20	15.4	23.4	20.2

Table 1: Multicasting timing (in ms)

Multicast method	Variable overhead	fixed overhead	Total overhead
Kernel-level	0.15	0.60	0.75
Push	0.32	0.60	0.92
User-level	0.57	0.60	1.17

Table 2: Incremental processing time per destination (in ms)

3.2 Multi RPC

The setup for this experiment is similar to the one used for multicasting (figure 3). The user-level program has to make a separate system call for each send and receive operation. The Push program needs only one system call. It sends the message to all destinations and collects the answers before returning to the user. A timeout mechanism is used to detect site failures.

Table 3 reports the results of this experiment. We did not implement a kernel-level version of multi RPC. The numbers in the first column are estimates that we obtained using the measurements observed in [BMR87]. For twenty destinations, we observe a 27% improvement over the user-level program and a 26% degradation from the kernel-level routine. This is better than the performance observed in the multicast experiment, where we had only a 15% improvement over the user-level program and a 31% degradation from the native kernel version. This is because each destination requires two system calls for the user-level implementation of multi RPC. Push is especially efficient when the user-level implementation of a service demands a heavy user-kernel interaction. The high overhead observed for one destination in the Push implementation is due to the extra complexity added by Push to the system call abstraction. Subsection 3.5 suggests ways to reduce this overhead. The

Number of destinations	kernel level SE	user level SE	Push
1	2.2	3.0	6.6
5	9.5	14.9	14.6
10	18.5	29.7	25.0
15	29.5	44.3	35.6
20	36.5	59.0	46.2

Table 3: Multi RPC timing (in ms)

multi RPC program can be easily modified to provide services that read/write data from/to different sites with one system call. Quorum formation can also be efficiently implemented using similar kernel-resident routines[Gif79].

3.3 Commitment Protocol

In Camelot [Spe86], the authors suggest that certain distributed transactions protocols can be added to the operating system to improve performance and to raise the level of the operating system interface. In database-oriented operating systems, commitment protocols can be added to the kernel. During transaction processing, the addresses of the participant sites can be registered. When the system is ready to commit the transaction, a single command in the database code will suffice. The performance is improved because of the reduced user-kernel interaction. The database system can also readily switch between alternative commitment protocols according to the demands of the system. Two-phase commit protocols are often used despite their blocking drawback [Ske82], because the message exchanges that take place during each phase impose a significant overhead on the system. The performance improvements provided by Push can make the implementation of three-phase commit protocols a practical solution to the blocking problems.

The two-phase commitment protocol used for this experiment is an extension of the multi RPC routine. The first phase is basically a multi RPC. In the second phase, the commitment decision is multicast to all participant sites. Since Push provides access to the file system, we can easily add logging operations to the protocol. Logging overhead has a significant impact on database performance, especially on transaction response time. Most of data I/O activity can be optimized by adequate caching policies. Writes to the log however, can not be delayed and have to be carried out before any commit decision is made. Push offers mechanisms to optimize those functions. Different schemas for interleaving communication, logging and computation can be readily tested with Push.

Table 4, shows commit times for different sets of participant sites. These times do not

Number of participants	kernel level SE	user level SE	Push
1	3.0	4.2	7.5
5	13.2	20.8	19.1
10	26.0	41.4	34.0
15	40.8	61.8	49.1
20	51.5	82.4	64.2

Table 4: Commit protocol timing (in ms)

include any disk activity. The user-level implementation of the two-phase commitment protocol demands three system calls per participant site plus the necessary logging activity, which may result in several additional system calls for writing to the disk log. The performance of the Push version is closer to that of the kernel-level version. For twenty sites, the performance is improved by 28% with respect to the user-level implementation and the degradation from the kernel-level implementation is only 24%.

3.4 File Copy

The response time of transaction processing depends on the performance of the underlying file system. The user interface presented by the file system may not be convenient for implementing transaction processing algorithms [Sto81]. We have written Push routines that extend the Unix file system to accommodate it to the demands of database systems. These routines use the file system primitives *creat, open, close, read, write* provided by the Push machine. Push routines can implement indexed access to file records, provide encryption capabilities, support recovery from crashes, etc. Since this is done inside the kernel, security and transparency are automatically provided.

The Push program in appendix C requires only one system call to copy a file, independent of its length. Table 5, shows the performance of that program. A similar, user-level facility produced slightly slower results. Currently, Push uses the standard Unix file system call interface. We are working on the implementation of a more efficient file system primitives for Push. In the future, Push programs will be able to avoid the overhead of copying data between the kernel input and output buffers, saving up to 20% of the time.

Bytes:	1K	4K	16K	64K	256K	1M	2M	4M	8M
Time:	6	11	33	125	596	2,504	18,282	42,616	92,099

Table 5: File copy times (in ms)

3.5 Performance Improvements to the Push Machine

Performance of the Push implementation can be improved in several ways. The general purpose memory allocator for the SunOS kernel is too inefficient, especially for small chunks of memory. We measured 500 μs for the allocation-deallocation of 50 bytes. We plan to have our own memory allocation scheme to avoid this overhead. The relative high start-up cost (highlighted by the cost of the services for a single destination) can be optimized by reducing the number of times Push has to cross the user/kernel boundary during input-output of parameters. Finally, the Push machine itself can be made more powerful to reduce the interpretation overhead (Push programs would consist of less instructions). For example, instead of the sequence *push a, push l, push m, send*, which is currently used to send the message *m* to network address *a*, we would have one instruction, namely *send m, l, a*.

4 Other Paradigms for Extensible Operating Systems

Several paradigms to achieve extensibility in operating systems have been proposed and implemented. They include parameterized operating systems, minimal kernels, synthesized code, streams, and the packet filter approach.

Monolithic operating systems offer a limited degree of flexibility. Configuration files and compilation or boot-time parameters are used by those systems to alleviate the problem. Digital Equipment Corporation's configuration expert system, XICON, can assist users in the customized configuration of a complete computing system [BM84]. To avoid overcrowding in the kernel, certain operating systems services have been implemented as user-level processes. These processes called *daemons*, run in close relation with the kernel. However, because all crucial information resides inside the kernel, performance and even consistency cannot be guaranteed. For example, in the context of Unix, the use of a daemon to implement routing protocols introduces inconsistencies between the views of the routing tables for the daemon and the kernel. Push programs running inside the kernel can avoid such inconsistencies by directly accessing kernel tables. Furthermore, the increased flexibility provided by Push can significantly reduce the size of these systems. The operating system would not have to be configured with all possible services.

Hoare proposed the small-kernel approach to operating systems [Hoa72]. Under this model, the kernel provides only basic services, i.e., process and memory management, and interprocess communication. On top of this infrastructure, a customized operating system can be built to support a given processing and hardware environment. His thesis is valid for time-sharing environments, where the basic task of the operating system is to share the computer resources among a variety of users. In this case, generalizing the operating system services to accommodate all potential uses of the system results in obtrusive, unreliable, and inefficient kernels.

In the last decade, several small-kernel operating systems have been proposed and implemented [Che84, YTR+87, DJA88, RAA+88]. Operating system services are provided as server processes. These servers can provide not only conventional operating system services such as file systems and network communication, but many other services for different applications. For example, we could have lock managers, atomicity controllers, consistency controllers to support distributed transaction processing. This approach is inappropriate for architectures with expensive context switches, since the kernel and the servers in the operating system are implemented in separate hardware protection domains. Switching between domains significantly increases the cost of the services. The operating system support demanded by large applications like distributed database management systems can be determined in advance and be included in the kernel. The resulting specialized kernel will provide better support for the implementation of reliable, high-performance database systems. Push can be used in a small kernel to supply operating system services without context switch overhead.

The Synthesis kernel suggests a solution that goes beyond the efficiency/power tradeoff that was mentioned above [PMI88]. This approach employs a monolithic kernel and uses several techniques to specialize the kernel code that executes specific requests. These techniques include the elimination of redundant computation and the collapsing of kernel layers. Synthesized code is reported to reduce the conventional execution path of some system calls by a factor of 10–20. This makes sense in general-purpose operating systems, where every user request has to be penalized by layers of code, that may be unnecessary for that specific request. For example, the Unix BSD model for interprocess communication, whose main goal is generality, results in an expensive sequence of procedure calls. Many of those procedure calls are irrelevant to individual messages [BMR87]. The Synthesis project also studied the problem of reducing the context switch overhead [MP89]. Their solution is based on additional hardware support. Push can be used to reduce context switch time along with Synthesis improvements in the performance of layered code, on systems without the special hardware support.

Streams increase the modularity and reusability of kernel code in the input-output subsystem [Rit84]. Streams try to eliminate the duplication of functionality existing in conventional device drivers. A stream is a two-way connection between a process and a device driver. Modules that process data flowing along this two-way path can be inserted and deleted dynamically, changing the behavior of the user interface. For instance, a user can create a stream between his process and a network device driver. Communication modules can then be added to that stream to implement a given suite of protocols. Currently, only kernel-resident stream modules can be pushed to and popped from a stream. Push offers increased flexibility by allowing users to write and push their own modules, once the initial raw stream has been created. Here, we see a synergism, produced by the cooperative use of streams and Push. New communication protocol suites can be implemented and tested using a stream connecting the user with the network interface. Modules written in Push can then implement the different layers of the protocol suite.

The packet filter presents another alternative to the efficiency/flexibility dilemma for network code implementation [MRA87]. The packet filter demultiplexes network packets according to rules specified by the users. These rules can be quite complex and can be changed dynamically. By running inside the kernel, the packet filter eliminates much of the context switch overhead incurred by user-level demultiplexers. At the same time, the overhead introduced by the interperter does not significantly affect the performance of network protocols when compared with native kernel code. The packet filter implementation supports evaluation of simple predicates. Push extends the technique to general purpose algorithms.

Extensions to the Unix file system have also been proposed in [BP88]. There, the additional file system services are implemented in user-level servers. The Unix kernel is modified to associate special processing requests with files. When a read, write, open, or close operation is invoked on such a file, the request is routed through a designated process, which may modify the interpretation of the request. For instance, an encrypted file system can be implemented transparently by modifying the read and write system calls to automatically encrypt and decrypt blocks of the file. Push extends [BP88] by providing the same functionality with enhanced performance and security. For instance, in the Push implementation of the encrypted file system encryption and decryption would be carried out entirely in the kernel, reducing the security risk.

5 Summary and Future Work

Push is a tool that allows database implementors to adjust the operating system functionality to their needs without sacrificing efficiency. For services that demand a constant interaction with the kernel, the performance advantages of Push over user–level implementations are clear. On the other hand, implementations of kernel code demand more effort than the corresponding implementations using Push. If performance is a real issue, and services have to be implemented inside the kernel, Push still can be used to test the services before the actual implementation takes place. The overhead in size and interpretation time introduced by Push are relatively small and their effects on the performance of an operating system service can be predicted with acceptable accuracy. We can determine the number of instructions executed by a Push program, and we have good estimates for the interpretation times of each Push instruction. This is important when using Push as an experimental tool to compare the performance of two or more potential kernel implementations of an operating system function.

We are building a simple yet powerful Push interface to basic operating system services already existing in the kernel. We want to extend the current implementation so that new operating system services implemented with Push can be tested in the context of the Raid distributed database system [BR89].

References

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Company, 1987.

[BM84] J. Bachant and J. McDermot. R1 revisited: Four years in the trenches. *AI Magazine*, 5(3):21–32, September 1984.

[BM89] Kenneth Birman and Keith Marzullo. ISIS and the META project. *Sun Technology*, pages 90–104, July 1989.

[BMR87] Bharat Bhargava, Tom Mueller, and John Riedl. Experimental analysis of layered Ethernet software. In *Proc of the ACM-IEEE Computer Society 1987 Fall Joint Computer Conference*, pages 559–568, Dallas, Texas, October 1987.

[BMRS89] Bharat Bhargava, Enrique Mafla, John Riedl, and Bradley Sauder. Implementation and measurements of an efficient communication facility for distributed database systems. In *Proc of the Fifth International Conference on Data Engineering*, Los Angeles, California, February 1989.

[BP88] Brian N. Bershad and C. Brian Pinkerton. Watchdogs: Extending the UNIX file system. In *Proc of the USENIX Winter Conference*, pages 267–275, Dallas, TX, February 1988.

[BR89] Bharat Bhargava and John Riedl. The Raid distributed database system. *IEEE Transactions on Software Engineering*, 15(6), June 1989.

[Che84] David R. Cheriton. The V kernel: A software base for distributed systems. *IEEE Software*, 1(2):19–42, April 1984.

[DJA88] Partha Dasgupta, Richard J. LeBlanc Jr., and William F. Appelbe. The CLOUDS distributed operating system: Functional description, implementation details and related work. In *Proc of the 8th Intl Conf on Distributed Computing Systems*, San Jose, CA, June 1988.

[Gif79] D. K. Gifford. Weighted voting for replicated data. In *Proc of the 7th Symposium on Operating Systems Principles*, pages 150–162, Asilomar, California, December 1979.

[Hoa72] C. A. R. Hoare. Operating systems: Their purpose, objectives, functions, and scope. In Hoare and Perrot, editors, *Operating System Techniques*, pages 11–25. Academic Press, 1972.

[MP89] Henry Massalin and Calton Pu. Fine-grain scheduling. In *Proc of the USENIX Workshop on Experiences with Distributed and Multiprocessor Systems*, pages 91–104, Fort Lauderdale, FL, October 1989.

[MRA87] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proc of the 11th ACM Symposium on Operating Systems Principles*, pages 39–51, Austin, TX, November 1987.

[PMI88] Calton Pu, Henry Massalin, and John Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, 1988.

[RAA+88] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. CHORUS distributed operating system. *Computing Systems*, 1(4):305–370, 1988.

[Rit84] D. M. Ritchie. A stream input–output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[SDE85] Michael Stonebraker, Deborah DuBourdieux, and William Edwards. Problems in supporting data base transactions in an operating system transaction manager. *Operating System Review*, 19(1):6–14, January 1985.

[Ske82] D. Skeen. Nonblocking commit protocols. In *Proc of the ACM SIGMOD Conference on Management of Data*, pages 133–147, Orlando, Florida, June 1982.

[Spe86] Alfred Z. Spector. Communication support in operating systems for distributed transactions. In *Networking in Open Systems*, pages 313–324. Springer Verlag, August 1986.

[Sto81] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981.

[YTR+87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proc of the 11th Symposium on Operating Systems Principles*, pages 63–76, Austin, TX, November 1987.

APPENDICES

A Summary of the Push Operations

Arguments in *italics* are from the stack. Other arguments are compiled into the instruction.

push	i	push i on the stack
pop	v	pop a value off the stack and assign it to variable v
dec	v	decrement the value of v by one
inc	v	increment the value of v by one
add	*a b*	push *a* + *b* on the stack
sub	*a b*	push *a* - *b* on the stack
jmp	l	jump to label l
jz	l	pop one element off the stack; jump to label l if zero
jnz	l	pop one element off the stack; jump to label l if not zero
alloc	*l* v	allocate *l* bytes to pointer v
free	*b*	free memory block *b*
copy	*a b l*	copy *l* bytes from *a* to *b*
compare	*a b l*	compare *l* bytes from addresses *a* and *b*
send	*m l a*	send *l* bytes from buffer *m* to network address *a*
recv	*m l a*	receive at most *l* bytes at address *m*
creat	*n m*	create a file with name *n* and mode *m*
open	*n f*	open the file *n* to R/W according to the flags *f*
close	*f*	close the file with file descriptor *f*
read	*f b l*	read *l* bytes from file *f* to buffer *b*
write	*f b l*	write *l* bytes from buffer *b* to file *f*
settimer	*s l*	set a timer for *s* seconds; if the timer expires jump to label *l*
stoptimer		disable a timer set earlier
treset		start timing
tprint		stop timing, place elapsed time on the stack
printi	v	print integer v
prints	*l* v	print *l* bytes, starting at address *v*
return		return to the user level

B Push Multicast Program

```
%Push multicast procedure

addrlen   def       6

addrs     in        address
addrcnt   in        integer
msg       in        address
msglen    in        integer

nxtaddr   var       address

          push      addrs      % nxtaddr = addrs
          pop       nxtaddr

loop      push      nxtaddr    % send (msg, msglen, nxtaddr)
          push      msglen
          push      msg
          send

          push      nxtaddr    % nxtaddr = nxtaddr + addrlen
          push      addrlen
          add
          pop       nxtaddr

          push      addrcnt    % addrcnt = addrcnt - 1
          dec
          dup
          pop       addrcnt

          jgt       loop       % if (addrcnt > 0) goto loop

          return
```

C Push File Copy Routine

```
BLEN    def     1024
READ    def     0
WRITE   def     1
pathr   in      address
pathw   in      address
buf     var     address
len     var     integer
fdr     var     integer
fdw     var     integer

        push    BLEN
        alloc   buf

        push    READ    %open source and destination files
        push    pathr
        open
        pop     fdr
        push    WRITE
        push    pathw
        open
        pop     fdw

l1      push    BLEN    %read from source file
        push    buf
        push    fdr
        read
        dup
        jle     l2
        push    buf     %write to destination file
        push    fdw
        write
        pop     len
        jmp     l1      %loop until end of source file

l2      push    fdr     %close files
        close
        push    fdw
        close

        return
```